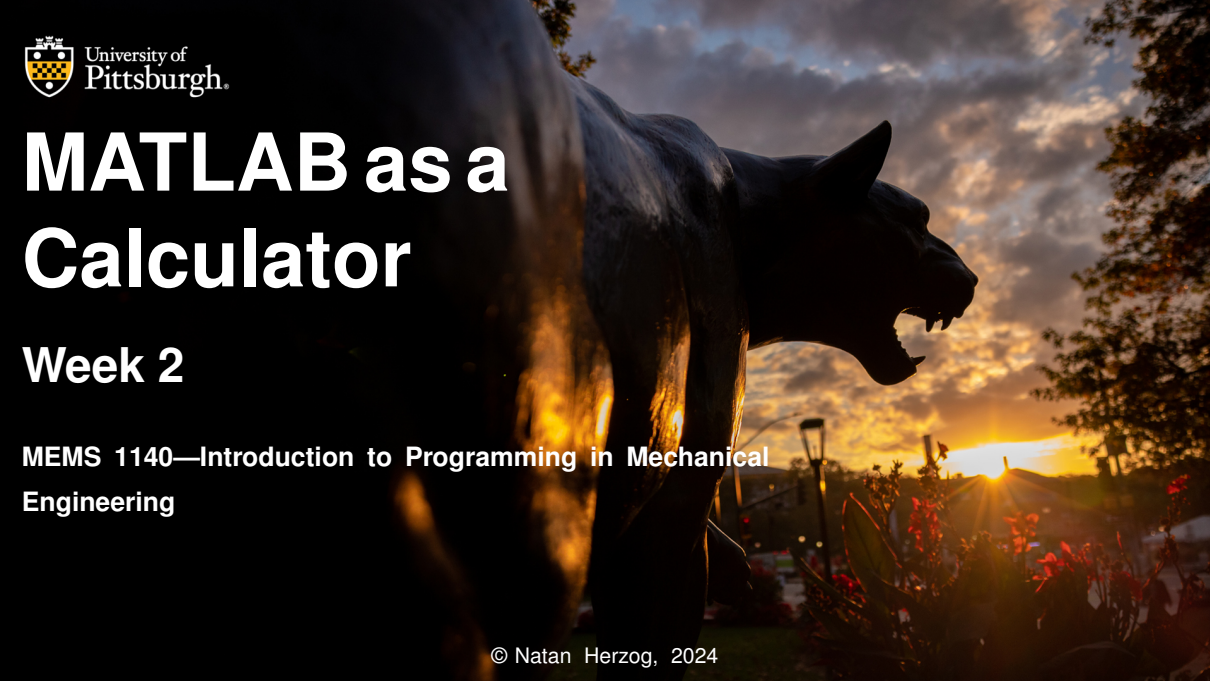


MATLAB as a Calculator

Week 2

MEMS 1140—Introduction to Programming in Mechanical Engineering

Learning Objectives (L.O.)

At the end of this lecture, you should understand/be able to:

- ❏ A simple calculator-type application for programming;
- ❏ Basic programming in engineering;
- ❏ Use the Command Window for basic math;
- ❏ Solve a simple engineering problem;
- ❏ Vector arithmetic in MATLAB.

Table of Contents (ToC)

1. Motivation for Programming as a Calculator
2. Relation to programming
3. Basic Math
4. Application to a Familiar Problem
5. Vector Arithmetic
6. Summary

1 – Motivation

Advanced topics in computing include:

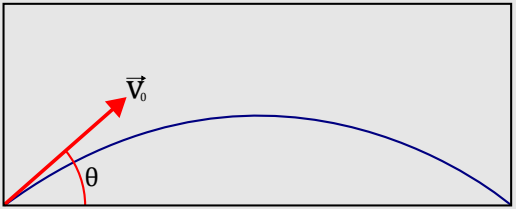
- numeric simulations
- machine learning
- quantum computing
- robotics

Basic computation is a natural prerequisite to all of these.

We'll consider a simple mechanics problem as an introduction.

1 – Simple Engineering Application

Ex: How far will a projectile travel given an initial velocity $v|_{t=0}$ and launch angle θ?



Don't worry, I will do the math.

You can focus on learning the code.

The solution for this utilizes just two kinematic equations.

1 – Kinematic Equations

Kinematic Equations

$$\underbrace{v_y = \left(v_y|_{t=0}\right) + a_y t}_{\text{1. solve for } t}$$

1 – Kinematic Equations

Kinematic Equations

$$\underbrace{v_y = \left(v_y|_{t=0}\right) + a_y t}_{\text{1. solve for } t} \qquad v_y|_{t=0} = \left(v|_{t=0}\right)\sin(\theta)$$

1 – Kinematic Equations

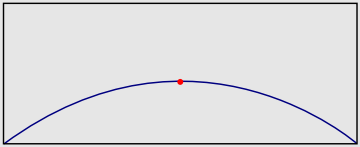
⇨ L.O.1
❑ L.O.2
❑ L.O.3
❑ L.O.4
❑ L.O.5

Kinematic Equations

$$\underbrace{v_y = \left(v_y|_{t=0}\right) + a_y t}_{\text{1. solve for } t}$$

$$v_y|_{t=0} = \left(v|_{t=0}\right)\sin(\theta)$$

$v_y = 0$ at the peak



1 – Kinematic Equations

Kinematic Equations

$$\underbrace{v_y = \left(v_y|_{t=0}\right) + a_y t}_{\text{1. solve for } t}$$

$$v_y|_{t=0} = \left(v|_{t=0}\right)\sin(\theta)$$

$v_y = 0$ at the peak

$a_y = -9.8\ \left[\mathrm{m/s^2}\right]$ is gravity

1 – Kinematic Equations

⇨ L.O.1
❑ L.O.2
❑ L.O.3
❑ L.O.4
❑ L.O.5

Kinematic Equations

$$\underbrace{v_y = \left(v_y|_{t=0}\right) + a_y t}_{\text{1. solve for } t}$$

$$\underbrace{\Delta x = \left(v_x|_{t=0}\right) t}_{\text{2. solve for } \Delta x}$$

1 – Kinematic Equations

⇨ L.O.1
❑ L.O.2
❑ L.O.3
❑ L.O.4
❑ L.O.5

Kinematic Equations

$$\underbrace{v_y = \left(v_y|_{t=0}\right) + a_y t}_{\text{1. solve for } t}$$

t is known from Step 1.

$$\underbrace{\Delta x = \left(v_x|_{t=0}\right) t}_{\text{2. solve for } \Delta x}$$

1 – Kinematic Equations

Kinematic Equations

$$\underbrace{v_y = \left(v_y|_{t=0}\right) + a_y t}_{\text{1. solve for } t}$$

$$\underbrace{\Delta x = \left(v_x|_{t=0}\right) t}_{\text{2. solve for } \Delta x}$$

t is known from Step 1.

$$v_x|_{t=0} = \left(v|_{t=0}\right)\cos(\theta)$$

2 – Relation to Programming

Both steps in solving this problem are just solving an equation.

This is the perfect application for a simple script.

After covering the basics of computation, we will circle back to this problem and solve it with a script.

3 – Addition & Subtraction

MATLAB is fantastic for doing simple math like a calculator.

For example, take the following arithmetic operations:

```
>> a = 5;                    % define variable a
.
.
.
.
.
.
```

Command Window

3 – Addition & Subtraction

MATLAB is fantastic for doing simple math like a calculator.

For example, take the following arithmetic operations:

```
>> a = 5;
>> b = a + 2                    % add 2 to the value of a
b =
     7
.
.
.
```

Command Window

3 – Addition & Subtraction

MATLAB is fantastic for doing simple math like a calculator.

For example, take the following arithmetic operations:

```
>> a = 5;
>> b = a + 2
b =
     7
>> c = a + b - 2              % add a and b and subtract 2
c =
     10
```

Command Window

3 – Multiplication, Division, Exponents

Multiplication, division, and exponents are similarly straightforward:

```
>> a = 5; b = 7; c = 10;     % define variables
.
.
.
.
.
.
```

Command Window

3 – Multiplication, Division, Exponents

Multiplication, division, and exponents are similarly straightforward:

```
>> a = 5; b = 7; c = 10;
>> d = a * b / c              % multiplication and division
d =
     3.5000
.
.
.
```

Command Window

3 – Multiplication, Division, Exponents

Multiplication, division, and exponents are similarly straightforward:

```
>> a = 5; b = 7; c = 10;
>> d = a * b / c
d =
    3.5000
>> e = a^2 * b / c^(1/3)     % introduce exponents
e =
    81.2278
```

Command Window

3 – Order of Operations

MATLAB also follows PEMDAS.

Just like in normal math, this governs the order in which operations will be computed.

Parentheses `(x)`

Exponent

Multiplication

Division

Addition

Subtraction

3 – Order of Operations

MATLAB also follows PEMDAS.

Just like in normal math, this governs the order in which operations will be computed.

Parentheses	`(x)`
Exponent	`a^b`
Multiplication	
Division	
Addition	
Subtraction	

3 – Order of Operations

MATLAB also follows PEMDAS.

Just like in normal math, this governs the order in which operations will be computed.

Parentheses	`(x)`
Exponent	`a^b`
Multiplication	`a*b`
Division	`a/b`
Addition	
Subtraction	

3 – Order of Operations

MATLAB also follows PEMDAS.

Just like in normal math, this governs the order in which operations will be computed.

Parentheses	`(x)`
Exponent	`a^b`
Multiplication	`a*b`
Division	`a/b`
Addition	`a+b`
Subtraction	`a-b`

3 – PEMDAS Example

Let's compare a couple of examples:

```
>> a + b / c          % no parentheses
ans =
     5.7000           % 5 + 7/10 = 5 + 0.7 = 5.7
.
.
.
.
.
.
```

Command Window

3 – PEMDAS Example

Let's compare a couple of examples:

```
>> a + b / c
ans =
     5.7000
>> (a + b) / c       % introduce parentheses
ans =
     1.2000          % (5 + 7) / 10 = 12 / 10 = 1.2
.
.
.
```

Command Window

3 – PEMDAS Example

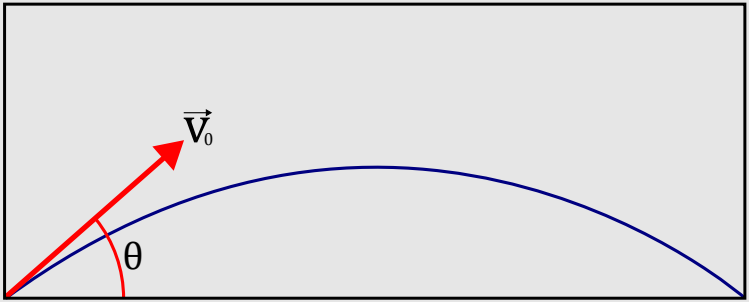
Let's compare a couple of examples:

```
>> a + b / c
ans =
     5.7000
>> (a + b) / c
ans =
     1.2000
>> (a^2 + b) / c     % include parentheses and introduce an exponent
ans =
     3.2000           % (5^2 + 7) / 10 = (25 + 7) / 10 = 32 / 10 = 3.2
```

Command Window

4 – Recall the Application

Armed with this knowledge — **Recall**: How far will a projectile travel given an initial velocity $v|_{t=0}$ and launch angle θ?



4 – Preparing Equations

Step 1

Solve for the time t at which the projectile reaches its peak.

$$\cancel{v_y}^{0} = \left(v_y|_{t=0}\right) + a_y t$$

$$t = -\frac{\left(v_y|_{t=0}\right)}{a_y} \quad \rightarrow \quad \underbrace{t = -\frac{\left(v|_{t=0}\right)\sin(\theta)}{-9.8}}_{\text{Step 1 in code}}$$

4 – Preparing Equations

Step 2

Solve for the distance that the projectile travels.

$$\underbrace{\Delta x = \left(v_x|_{t=0}\right)(2t)}_{\text{Step 2 in code}}$$

Recall that t was the time *to the peak*, which is only half of the total flight time.

4 – Preparing Equations

This gives us the following two equations:

$$\underbrace{t = -\frac{\left(v|_{t=0}\right)\sin(\theta)}{-9.8}}_{\text{Step 1}} \qquad \underbrace{\Delta x = \left(v_x|_{t=0}\right)(2t)}_{\text{Step 2}}$$

Both of these equations can be very easily solved in a script.

4 – Defining Constants

1. Define constants ($v|_{t=0} = 10\,[\mathrm{m/s}]$ and $\theta = 30°$):

```
initial_velocity = 10;                    % [m/s]
.
.
.
.
.
.
.
```

MATLAB Script (.m)

4 – Defining Constants

1. Define constants ($v|_{t=0} = 10\,[\mathrm{m/s}]$ and $\theta = 30^\circ$):

```
initial_velocity = 10;                    % [m/s]
launch_angle = 30;                        % degrees
.
.
.
.
.
.
```

MATLAB Script (.m)

4 – Defining Constants

1. Define constants ($v|_{t=0} = 10\,[\mathrm{m/s}]$ and $\theta = 30^\circ$):

```
initial_velocity = 10;                % [m/s]
launch_angle = 30;                    % degrees
gravity = -9.8;                       % [m/s^2]
.
.
.
.
.
```

MATLAB Script (.m)

4 – Solving Initial Velocities

2. Solve for initial velocity components:

```
initial_velocity = 10;                                  % [m/s]
launch_angle = 30;                                      % degrees
gravity = -9.8;                                         % [m/s^2]

v_y_0 = initial_velocity * sind(launch_angle);          % v_y @ t=0
.
.
.
```

MATLAB Script (.m)

4 – Solving Initial Velocities

2. Solve for initial velocity components:

```
initial_velocity = 10;                                  % [m/s]
launch_angle = 30;                                      % degrees
gravity = -9.8;                                         % [m/s^2]

v_y_0 = initial_velocity * sind(launch_angle);          % v_y @ t=0
v_x_0 = initial_velocity * cosd(launch_angle);          % v_x @ t=0
.
.
```

MATLAB Script (.m)

4 – Solving for Time to Peak

3. Solve for the time to the peak, t:

```
initial_velocity = 10;                              % [m/s]
launch_angle = 30;                                  % degrees
gravity = -9.8;                                     % [m/s^2]
v_y_0 = initial_velocity * sind(launch_angle);      % v_y @ t=0
v_x_0 = initial_velocity * cosd(launch_angle);      % v_x @ t=0

t_peak = -(v_y_0 / gravity)                         % time to peak
.
```

MATLAB Script (.m)

The projectile reaches its peak at $t = 0.5102$ [s].

4 – Solving for Distance Traveled

Solve for Δx:

```
initial_velocity = 10;                                  % [m/s]
launch_angle = 30;                                      % degrees
gravity = -9.8;                                         % [m/s^2]
v_y_0 = initial_velocity * sind(launch_angle);          % v_y @ t=0
v_x_0 = initial_velocity * cosd(launch_angle);          % v_x @ t=0
t_peak = -(v_y_0 / gravity)                             % time to peak

x = v_x_0 * (2*t_peak)                                  % distance traveled
```

MATLAB Script (.m)

The projectile reaches 8.8370 [m].

5 – Vectors

In addition to basic computation, it is helpful to have an understanding of how to work with vectors in MATLAB.

In mechanical engineering, vectors are used to describe:

- Force application;
- Momentum;
- Fluid Velocity fields;
- etc.

5 – Vectors in MATLAB

In math, we represent cartesian vectors with components along each direction:

$$\vec{v} = [v_x,\ v_y,\ v_z]$$

In code, we define an array to do the same:

```
v = [v_x, v_y, v_z];  % a 1x3 array
```

5 – Unit Vectors

The most basic vectors are the three unit direction vectors:

$$\hat{\imath} = [1, 0, 0] \quad \hat{\jmath} = [0, 1, 0] \quad \hat{k} = [0, 0, 1]$$

These can be written in code as follows:

```
i_hat = [1, 0, 0];  % x-direction vector
.
.
```

MATLAB Script (.m)

5 – Unit Vectors

The most basic vectors are the three unit direction vectors:

$$\hat{\imath} = [1, 0, 0] \quad \hat{\jmath} = [0, 1, 0] \quad \hat{k} = [0, 0, 1]$$

These can be written in code as follows:

```
i_hat = [1, 0, 0];  % x-direction vector
j_hat = [0, 1, 0];  % y-direction vector
.
```

MATLAB Script (.m)

5 – Unit Vectors

The most basic vectors are the three unit direction vectors:

$$\hat{\imath} = [1, 0, 0] \quad \hat{\jmath} = [0, 1, 0] \quad \hat{k} = [0, 0, 1]$$

These can be written in code as follows:

```
i_hat = [1, 0, 0];  % x-direction vector
j_hat = [0, 1, 0];  % y-direction vector
k_hat = [0, 0, 1];  % z-direction vector
```

MATLAB Script (.m)

5 – Adding Unit Vectors

To expand from here, multiples of the unit vectors can be added together to construct any 3-dimensional vector.

```
i_hat = [1, 0, 0];          % x-direction vector
j_hat = [0, 1, 0];          % y-direction vector
k_hat = [0, 0, 1];          % z-direction vector
.
```

MATLAB Script (.m)

5 – Adding Unit Vectors

To expand from here, multiples of the unit vectors can be added together to construct any 3-dimensional vector.

```
i_hat = [1, 0, 0];                  % x-direction vector
j_hat = [0, 1, 0];                  % y-direction vector
k_hat = [0, 0, 1];                  % z-direction vector
v = 3*i_hat + 2*j_hat + k_hat       % new vector v
```

MATLAB Script (.m)

```
v = [3, 2, 1]
```

Command Window Output

5 – Adding non-unit Vectors

Similarly, non-unit vectors can also be added together:

```
u = [1, 2, 3];
.
.
```

MATLAB Script (.m)

5 – Adding non-unit Vectors

Similarly, non-unit vectors can also be added together:

```
u = [1, 2, 3];
v = [3, 2, 1];
.
```

MATLAB Script (.m)

5 – Adding non-unit Vectors

Similarly, non-unit vectors can also be added together:

```
u = [1, 2, 3];
v = [3, 2, 1];
w = u + v
```

MATLAB Script (.m)

Each corresponding component is added together, resulting in:

```
w = [4, 4, 4]
```

Command Window Output

5 – Can Vectors be Multiplied?

Attempting to multiply vectors **u** and **v** using the * operator will cause an error.

```
u = [1, 2, 3]; v = [3, 2, 1];
.
```

5 – Can Vectors be Multiplied?

Attempting to multiply vectors **u** and **v** using the * operator will cause an error.

```
u = [1, 2, 3]; v = [3, 2, 1];
w = u * v
```

```
Error using *
Incorrect dimensions for matrix multiplication.
.
.
.
```

Error Message

5 – Can Vectors be Multiplied?

Attempting to multiply vectors **u** and **v**, as defined previously, using the * operator cause an error:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = u * v
```

```
Error using *
Incorrect dimensions for matrix multiplication. Check that the number of
columns in the first matrix matches the number of rows in the second
matrix.
.
```

Error Message

5 – Can Vectors be Multiplied?

Attempting to multiply vectors **u** and **v** using the * operator will cause an error:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = u * v
```

```
Error using *
Incorrect dimensions for matrix multiplication. Check that the number of
columns in the first matrix matches the number of rows in the second
matrix. To operate on each element of the matrix individually, use TIMES
(.*) for elementwise multiplication.
```

Error Message

5 – Linear Algebra Refresher

As the error message explains:

Arrays of size **`MxN`** and **`AxB`** can be multiplied when **`N=A`**.

Two *vectors* can be multiplied, as in the dot product — also known as the inner product — as follows:

$$\begin{bmatrix} a_{11} & a_{12} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = (a_{11}b_{11} + a_{12}b_{21})$$

5 – Vector Inner Product

Given that **u** and **v** are both defined as row vectors, using the * operator for the inner product requires transposing **v**:

```
u = [1, 2, 3]; v = [3, 2, 1];
.
```

5 – Vector Inner Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the inner product requires transposing `v`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = u *
```

5 – Vector Inner Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the inner product requires transposing `v`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = u * transpose(v)
```

```
w =
     10        % (1*3) + (2*2) + (3*1)
```

Command Window Output

For vectors, this produces a single scalar value, as expected.

5 – Vector Outer Product

In contrast, multiplying a column vector by a row vector, known as the outer product, produces a matrix:

$$\begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} \\ a_{21}b_{11} & a_{21}b_{12} \end{bmatrix}$$

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
.
```

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u)
```

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     .     .      % (1*3)
     .     .     .
     .     .     .
```

Command Window Output

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     2     .       % (1*3)    (1*2)
     .     .     .
     .     .     .
```

Command Window Output

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     2     1      % (1*3)   (1*2)   (1*1)
     .     .     .
     .     .     .
```

Command Window Output

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     2     1      % (1*3)   (1*2)   (1*1)
     6     .     .      % (2*3)
     .     .     .
```

Command Window Output

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     2     1      % (1*3)   (1*2)   (1*1)
     6     4     .      % (2*3)   (2*2)
     .     .     .
```

Command Window Output

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     2     1      % (1*3)   (1*2)   (1*1)
     6     4     2      % (2*3)   (2*2)   (2*1)
     .     .     .
```

Command Window Output

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     2     1      % (1*3)   (1*2)   (1*1)
     6     4     2      % (2*3)   (2*2)   (2*1)
     9     .     .      % (3*3)
```

Command Window Output

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     2     1      % (1*3)   (1*2)   (1*1)
     6     4     2      % (2*3)   (2*2)   (2*1)
     9     6     .      % (3*3)   (3*2)
```

Command Window Output

5 – Vector Outer Product

Given that `u` and `v` are both defined as row vectors, using the `*` operator for the outer product requires transposing `u`:

```
u = [1, 2, 3]; v = [3, 2, 1];
w = transpose(u) * v
```

```
w =
     3     2     1     % (1*3)   (1*2)   (1*1)
     6     4     2     % (2*3)   (2*2)   (2*1)
     9     6     3     % (3*3)   (3*2)   (3*1)
```

Command Window Output

5 – Can Vectors be Multiplied?

So ... can vectors be multiplied together?

Yes, vectors in MATLAB *can* be multiplied together.

But their dimensions must be an appropriate match and they must be oriented in the correct way.

6 – Summary

This lecture covered:

- ✓ A simple arithmetic example for programming

  Simple physics problems that only require a series of algebraic calculations are perfect for writing a little bit of code.

- ✓ How to relate engineering problems to programming

  Recognizing that a problem only involves basic computations should make programming a more attractive approach to solving it.

6 – Summary

✓ How to use the Command Window for basic math

The Command Window is easily accessible for executing one-line math operations, such as basic arithmetic. Each operation can be entered and executed in order.

✓ How to solve the example provided

The series of algebraic calculations required for the basic application can be executed as a script to progress through solving the problem.

6 – Summary

✓ How to compute vector arithmetic in MATLAB

Vectors can be written as `1x3` arrays and can be added and subtracted from each other on an element-wise basis, just like in math. They can also be multiplied to perform the inner and outer products, as long as their dimensions and orientations are appropriate.